DATE: March 29, 1976

Programming and Engineering Staff

FROM: M. L. Grubin

SUBJECT: P-4CC PROCESS EXCHANGE AND NEW PROTOCOLS

I. Process Exchange

A. Data Bases
   1. Ready List
   2. WAIT Lists
   3. Process Control Block (PCB)

B. Instruction Primitives
   1. WAIT
   2. NOTIFY

C. Dispatcher and Register File Management
   1. Ready List Maintenance
   2. Register Set Assignment
   3. Fetch Cycle Trap

II. Traps, Interrupts, Faults, Checks

A. External Interrupts
   1. Operation
   2. Special Instructions (IRTN, INOTIFY)

B. Faults
   1. Data Bases
   2. CALF
   3. Fault Handler

C. Checks

III. Register Files

IV. Control Panel

V. CP Timer
The Process Exchange mechanism is composed of three data bases and two basic instruction primitives. The data bases are the ready list, wait lists, and Process Control Blocks (PCB). The basic instruction primitives are WAIT and NOTIFY. In addition, there is an independent mechanism for controlling the usage of two register sets which is related to, but not part of, the ready list data base.

A. Data Bases

1. Ready List

The ready list is a two-dimensional list structure used for priority scheduling and dispatching of processes. The entire ready list data base (excluding live registers) and all PCB's are contained in a single segment. The segment number of this segment is contained in a 16-bit register called OWNERH. Within the segment, all pointers and addresses (except fault vectors and wait list pointers) are 16-bit word number quantities.

The two-dimensionality of the ready list is achieved with a Linear array of list headers for each priority level composed of a Beginning of List (BOL) pointer and an End of List (EOL) pointer.

Each pointer is the 16-bit word number address of a PCB (in the same segment as the ready list). The PCB's on each priority level list are forward-threaded through a 16-bit link word, and as many PCB's as desired can be threaded together on each priority level to form the ready list. A process' priority level is both determined by and encoded as the address of a BOL pointer in the ready list. Priority order is determined by arithmetic comparison, i.e., smaller numbers (addresses) are higher priorities. As a result, priority level list headers must be allocated in contiguous memory at system startup time.

The end of the ready list is determined by a EOL containing a 1 (PCB addresses must be even). An empty level is indicated by a EOL containing 0. Figure 1 is a picture of the ready list structure. The 32-bit registers PPA (Pointer to Process A) and PPP (Pointer to Process B) are a speed-up mechanism for locating the next process to dispatch. PPA always contains both the level (BOL pointer) and PCB address (designated level A and PCBA) of the currently active process. PPA points to the NEXT process to be run when process A 'goes away'. PPA not only points to the currently active process, but, by definition, level A is the highest level in the system. It is possible for PPA and PPP to be 'invalid'. This condition is indicated by a PCB address of 0. It is important NOT to disturb the level portions, especially level A since, even if invalid, level A indicates the highest level that WAS in the system and therefore determines where in the ready list to begin a scan, if necessary (PPP invalid), for the next process to run. In a completely idle system, both PPA and PPP will be invalid and, upon completion of the ready list scan, the u-code will go into a 'wait for interrupt' loop.
It is important to notice that there is no word number pointer to the first priority level in the ready list. The ready list allocator, which starts the process exchange mechanism, knows where the list begins and, during execution, level A (in PPA) will always point to either the highest level currently in the system or the last known highest level and, hence, acts as an effective ready list begin pointer. In addition, level 8 will always be higher than the second level to run. That is, a PCB can never be on a level higher than level 8 unless it is the only PCB higher than level 8 (i.e., level A).

Two "queuing" algorithms will be implemented for the ready lists, either FIFO or LIFO queuing.

2. WAIT Lists

Every PCB in the system will always be somewhere. If it is not on the ready list, then, by definition, it will be on a wait list. A wait list is defined by a 32-bit semaphore consisting of a 16-bit counter (C) and a 16-bit word number SOL pointer. Since the ready list and all PCB's reside in one segment (OWNERH), and only PCB's go onto wait lists, a segment number is not needed in the semaphore. However, semaphores themselves can be anywhere and, in general, are NOT in the PCS segment. The structure of a wait list is shown in Figure 2. Notice that the last block on the wait list contains a 0 link word as also that the semaphore contains only a SOL pointer.

The 'queuing' algorithm for wait lists is process priority queuing. That is, the priority level of a PCB will determine where on the wait list the PCB will be queued. For PCB's of equal priority, the algorithm becomes FIFO.

3. Process Control Block (PCB)

The contents of the PCB are shown in Figure 3. The PCB can be broken into the following logical sections which are ordered as shown:

a. Control

   0 - level (pointer to SOL in ready list)
   1 - link (pointer to next PCB or 0)
   2-3 - SN/AN of Wait List this block is currently on (SN=0 indicates on ready list)
   4 - abort flags used to generate Process Fault when PCB is dispatched.
      Current bit assignments 1-15: MEZ
      16: process interval timer overflow
   5,7 - reserved

b. Process State

   8-9 - Process elapsed timers (must be maintained by software that resets the interval timer)
10-12 - DTAR2 and DTAR3 (never saved, always restored)
14 - Process Interval Timer with 1.024 msec resolution
15 - Reserved
16 - Save mask - used to avoid saving and restoring
registers = 0
 Bits 1-8: GR0-GR7 (2 words each)
  9-12: FP0-FP1 (4 registers, 2 words each)
  13-16: Base Registers (PA,SB,LB,XB)

17 - Keys
18,33 - GR0-GR7
34,41 - FP0-FP1
42-49 - Base Registers (PA,SB,LB,XB)

Note that although all the registers are assigned locations
within the PCB, only non-zero registers will actually be saved
which results in a compacted list which can only be determined
by the bits in the save mask. In general, the saved registers
(those not equal to 0) will be between words 18 and 49. The
order of the registers, however, is fixed as above.

c. Fault (See section on Faults for a description of the use of
this portion of the PCB)
50-59 - Fault Vectors: SN/YN pointers to fault tables for
Ring C, Ring 1, Page Fault and Ring 3
fault handlers
60,62 - Concealed Fault Stack Header
63... - Concealed Stack - 6 word entries. (This stack need
not start at word 63).

B. Instruction Primitives
There are two basic instruction primitives for the process exchange
mechanism: NOTIFY and WAIT. In addition, NOTIFY has two variants.
These instructions, similar to Dijkstra's P and V operators, are
essentially "interlock" mechanisms. These instructions are three-word
(48-bit) "instructions" as follows:

Instruction (16-bit universal generic):
32-bit pointer to semaphore address

As suggested by the names, WAIT is used to wait for an event (CP, time,
unit record device available, whatever) and NOTIFY is used to signal
that an event has occurred. In particular, a WAIT is used to wait for
a NOTIFY and a NOTIFY is used to alert a process which is waiting.

Coordination is achieved by means of a semaphore containing a counter
and a BOL pointer. The semaphore and the PCB's waiting for the event
of that semaphore constitute a wait list. The counter, if greater than
0, indicates the number of PCB's on the wait list. If negative, it
indicates the number of processes that can obtain the resource.
Semaphores fall into two categories: public and private. A public
semaphore is used to coordinate several processes together or control a
system resource. Private semaphores are used by a single process to
coordinate its own activities. For example, if a disk request is made, a process will wait on a private semaphore for the disk operation to complete. The disk process will then notify the semaphore upon completion. The distinguishing characteristics of a private semaphore is that only 1 PCB can ever be on that wait list. A public semaphore can have many different PCB's on its wait list.

1. WAIT

The operation of wait is as follows: the semaphore counter is incremented and, if greater than 0, (resource not available/event has not occurred), the PCB is removed from the ready list and added to the specified wait list. If the counter is less than or equal to 0, the process continues. If the PCB is put on the wait list, the general registers are NOT saved and the register set is made available. Therefore, a process can NEVER depend on the general registers being intact after a WAIT. In fact, from the point of view of an executing process, a WAIT appears as a NOP which destroys the registers. In addition, WAIT will turn off the process timer. Figure 4 is a detailed flow chart of the WAIT instruction.

2. NOTIFY

The instructions differ ONLY in the ready list queuing algorithm used. The operation of NOTIFY is as follows: the semaphore counter is decremented and the notifying process continues. If the counter is less than 0, no action is taken, but if greater than or equal to 0, a PCB is removed from the top of the wait list and added to the ready list. No explicit action is ever taken on the notifying process, only the notified semaphore. If a notified process is of higher priority than the notifying process, the latter will be effectively 'interrupted', but will remain on the ready list. Figure 5 is a detailed flow chart of the NOTIFY instruction.

C. Dispatcher and Register File Management

The dispatcher is the root of the process exchange mechanism and is responsible for determining the next process to run (be dispatched), and assigning that process a register set. There is considerable overlap with NOTIFY and WAIT in functionality related to maintaining the ready list. For example, both NOTIFY and WAIT update PPA and PPB as appropriate, but the dispatcher scans the ready list if PPA is valid. Register file management, including any necessary saves and restores, are the sole province of the dispatcher. Figures 6 and 7 are detailed flow charts of the dispatcher.
1. Ready List Maintenance

Upon entry, the dispatcher first asks if PPA is valid. If it is, the process is assigned a register set and dispatched. If PPA is not valid, a scan of the ready list is initiated. If a PCP is found, PPA is adjusted and the process dispatched. If the ready list is empty, the dispatcher idles. Whenever a process is dispatched the process timer is turned on.

2. Register Set Assignment

In each register set, a register, designated OWNER, contains a pointer to the PCB of the process which owns the set. OWNER is a full 32-bit pointer and OWNERH is used throughout the system to determine the segment number of the ready list and PCB's. Obviously, OWNERH must be the same in both register sets. In addition, the low order bit of the keys register (KEYSH) is used to indicate whether the register set is available. The bit is called the Save Done bit and, if set, indicates that the register set and its copy in the owner's PCB are identical (a save has been done). This bit is set by the save routine (called from WAIT or the dispatcher) and reset when a process is dispatched. Whether a register set is available (SD=1) or not, it is always owned. Therefore, if a process goes away (either as a result of a WAIT or the notification of a higher level process) and comes back again immediately and, if that process still owns the register set, a restore operation is not necessary. If a register set switch is necessary, the process timer is turned off. The details of selecting which register set to assign to a process being dispatched is shown on the right of Figure 6. The dispatcher is the only code which switches register sets.

3. Fetch Cycle Trap

At various points in the dispatcher (indicated by I on the flow chart) a check for interrupt pending (fetch cycle trap) is made. As a result, interrupts can occur either in the fetch cycle or in the dispatcher. The possible Fetch Cycle traps are:

1. External Interrupt (See Part II-A)
2. CP-timer increment and possible overflow (See Part V)
3. Power Failure (See Part II-C)
4. Halt switch on control panel (See Part IV)
5. End-of-Instruction Trap

The end-of-instruction trap occurs either from an ECC corrected error or from a missing memory module, memory parity, or machine check during I/O. In all cases, if the check handling software returns (via LPSW instruction), the possible destinations are either the fetch cycle or the dispatcher (PSW in PSW not a real program counter). In order to guarantee the proper destination, bit 15 of the keys (KEYSH) is used to indicate if the trap was detected by the dispatcher (bit 15 = 1).
This bit is set by the dispatcher upon detecting a trap and is reset
in a process is actually dispatched (return to fetch cycle).

II. TRAPS/ INTERRUPTS/ FAULTS/ CHECKS

Four words used frequently are 'trap', 'interrupt' (or 'external
interrupt'), 'fault', and 'check'. The meanings of these terms are
carefully distinguished for the P-400/500. Software breaks in
execution are divided into three main categories referred to as
'interrupts', 'faults', and 'checks'. The word 'trap' on the other
hand, refers to a break in execution flow on the u-code level.

Traps can occur for many reasons, not all of which cause software
visible action, and are always processed on the u-code level. Some
traps may directly or indirectly cause breaks in software execution,
but not all software breaks are the result of a trap.

On the PRIME 300, interrupts, faults, and checks used the same protocol
to get to their respective software handlers, namely they caused a
vector through a dedicated sector 0 location (JST* vector). On the
P-400/500, when process exchange mode is enabled, the three categories
use different protocols both from the P-300 and each other. Roughly,
the three terms are used to describe:

1. Interrupt - a signal has been received from a device in the
   external world (including clocks) indicating that
   the device either needs to be serviced or has
   completed an operation. In general, an interrupt
   is not the result of an operation initiated by the
   currently executing software and will not be
   processed by that software (though, of course, it
   may).

2. Fault - a condition has been detected that requires
   software intervention as a direct result of the
   currently executing software. In general, faults
   can be handled by the current software, though in
   many cases common supervisor code within the
   current process handles the fault. Also, in
   general, an external device in the real world is
   not directly involved in either the cause or cure
   of a fault condition. Often, however, external
devices are involved indirectly as, for example,
in performing a page turn operation as a result of
a page fault.

3. Check - an internal CP consistency problem has been
detected which requires software intervention.
The condition could be either an integrity
violation, reference to a memory module which does
not exist, or a power failure. By contrast, a
reference to a page which is not resident or an
arithmetic operation which causes an exception is
a FAULT condition.
A. External Interrupts

1. Operation

External Interrupts operate in either of two modes depending upon whether process exchange is turned on. If process exchange is off, all interrupts are treated as P-3GC interrupts. In all cases, except memory increment, the address presented by the controller (or '63 if in standard interrupt mode) is used as the address in segment 0 of a 16-bit vector. This vector, in turn, points to interrupt response code (IRC), also in segment 0, which is entered via a simulated JST through the vector. Thus, the current P-counter (RPL) is stored in (vector) and execution begins at location (vector) +1 with interrupts inhibited but with no other keys or modals changed. If in vectored interrupt mode, it is the responsibility of the software to do a CAI. In either mode, the full RP is saved in the register PSWBP.

If process exchange mode is on, an entirely different mechanism operates. In all cases, except memory increment, the address presented by the controller is used as a 16-bit word number offset into the interrupt segment (I/O). This segment is guaranteed to be in memory, but STL misses may occur. The current PB (actually RP) and KEYS (keys and modals) are saved in the u-code scratch registers PSWBP and PSWKEYS. The machine is then inhibited and the IRC begins execution in 64V mode. It is the responsibility of the IRC to issue a CAI. It is important to note that the IRC in the interrupt segment does not belong to any process. PPA points to the PCB of the interrupted process and, in fact, no PCB exists for the IRC. Also, except for PB and KEYS, no registers are saved. In fact, even PSWBP and PSWKEYS are in the register file and not in memory. As a result, the IRC cannot do an enable and must return to the process exchange mechanism (i.e., the dispatcher) as soon as possible. Because of all these restrictions on what the immediate IRC can do, as well as the fact that it does not belong to any process, it is referred to as phantom interrupt code. Unless the job of servicing an interrupt is very simple, phantom interrupt code can do little more than turn off the controller's interrupt mask, issue a CAI, and NOTIFY the real IRC.

A memory increment interrupt is handled the same regardless of the state of process exchange. The address presented by the controller is used as the 16-bit word number in segment 0 (I/O segment) of a 16-bit memory cell to be incremented. If the counter does not overflow (-1->0), the u-code simply returns. With process exchange off, the return is always to the fetch cycle. With process exchange on, the return is either to the fetch cycle or the dispatcher, depending upon where the interrupt was detected. When detecting an interrupt, the dispatcher always insures that RP=PB and that all live keys=KEYS. If memory increment returns, it does so to the top of the dispatcher without having touched PB or KEYS. In this way, memory increment is guaranteed not to destroy any vital information needed by the dispatcher. If the memory cell counter does overflow, an End-of-Range interrupt is generated and then memory increment returns.
sequent EOR interrupt will then be treated like any other external interrupt. Figure 8 is a detailed flow chart of the external interrupt.

2. Special Instructions (IRTN, INOTIFY)

Phantom interrupt code has two options for the actions it can take. If the servicing required by the interrupt is very simple, phantom code can completely process the interrupt and return to the dispatcher. If the servicing required is more complex, the phantom code must turn off the controller’s interrupt mask and NOTIFY the remainder of the IRC. In the first case, PB and KEYS must be restored from PSWPS and PSWKEYS and then the dispatcher must be entered directly. Since PB cannot be restored in phantom code (the F-counter will point to the instruction in phantom code) and the dispatcher cannot be entered directly (no such instruction exists), the special instruction, IRTN, a 16-bit generic, is executed to perform these functions. After entering the dispatcher via an IRTN, the dispatcher does not know that an interrupt occurred.

In order to NOTIFY a process, phantom code must ensure that PB and KEYS are restored before issuing the NOTIFY. The special instruction, INOTIFY, performs the restore and then does the NOTIFY. As NOTIFY, INOTIFY is a three-word generic with two flavors, INOTIFYE and INOTIFYE, the beginning of list option has bit 16=1 and the end of list option has bit 16=0 in the opcode.

Phantom Interrupt code can issue a CAI in one of two ways. Either an explicit CAI instruction may be issued or the IRTN/INOTIFY instructions can issue it. Bit 15 of the IRTN/INOTIFY instructions is interpreted as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Bit 15</th>
<th>16</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>INEC</td>
<td>1</td>
<td>0</td>
<td>End + CAI</td>
</tr>
<tr>
<td>INEN</td>
<td>0</td>
<td>0</td>
<td>End + no CAI</td>
</tr>
<tr>
<td>INBC</td>
<td>1</td>
<td>1</td>
<td>Beginning + CAI</td>
</tr>
<tr>
<td>INBN</td>
<td>0</td>
<td>1</td>
<td>Beginning + no CAI</td>
</tr>
</tbody>
</table>

Figure 9 is a detailed flow chart of the IRTN and INOTIFY instructions.

Faults

Faults are CPU events which are synchronous with and, in a loose sense, caused by software. Eleven fault classes have been defined for the P-400. Several of these classes are further subdivided into distinct types. Of the eleven, three are completely new for the P-400 and, of
the other eight-three have expanded meaning when in P-400 mode. The eleven fault classes and their meanings are:

<table>
<thead>
<tr>
<th>Fault</th>
<th>P-400</th>
<th>P-300</th>
</tr>
</thead>
<tbody>
<tr>
<td>RXM</td>
<td>Restrict mode violation</td>
<td>same</td>
</tr>
<tr>
<td>Process</td>
<td>Abort flags word <em>HE</em> 0</td>
<td>N.A.</td>
</tr>
<tr>
<td>Page</td>
<td>Page Fault (Page not in memory)</td>
<td>same</td>
</tr>
<tr>
<td>SVC</td>
<td>N.A.</td>
<td>Supervisor Call</td>
</tr>
<tr>
<td>UII</td>
<td>Unimplemented instruction</td>
<td>same</td>
</tr>
<tr>
<td>ILL</td>
<td>Illegal instruction</td>
<td>same</td>
</tr>
<tr>
<td>Access</td>
<td>Violation of segment access</td>
<td>Page write violation</td>
</tr>
<tr>
<td></td>
<td>rights</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>All FLEX + IEX (Integer Exception)</td>
<td>FLEX</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack overflow/underflow</td>
<td>Procedure Stack (S-Reg)</td>
</tr>
<tr>
<td>Segment</td>
<td>1: Segment # too big</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>2: Missing segment (SDW fault bit set)</td>
<td>N.A.</td>
</tr>
<tr>
<td>Pointer</td>
<td>Fault bit in pointer set</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

The fault handling mechanism consists of two data bases and the CAL instruction. The u-code is in turn divided into a set of 'front-ends' for each fault class and a common fault handler.

1. Data Bases

The fault data bases consist of the fault vectors and concealed stack in the PCB and the fault tables pointed to by the PCB vectors. Figure 1C shows these data bases as well as the mapping of P-300 faults to P-400 faults. Also shown in this figure is the differential action taken according to fault class (e.g. what ring to process the fault in) and the set up the u-code 'front-end' must do before going to the common fault handler.

The underlying philosophy of the four fault vectors is that while some faults may need to be processed by ring 0 code, others may be adequately handled in the current ring of the faulting process. The vectors are in the PCB to allow different processes to have different fault handlers. For example, process A may wish to use a system fault routine to handle pointer faults (dynamic linker) while process B may wish to define its own algorithms for resolving pointer faults. Notice that it is always possible for a 'current ring' fault handler to call a ring 0 procedure if the need arises. Note also that page fault has its own vector despite the fact that ring 0 is entered. For the special case of page fault, only a single system-wide processor will be used and all PCB page fault vectors will point to the same place.

The concealed stack, also in the PCB, is used to allow fault on fault
onditions. For example, it is quite possible to get a segment fault on processing a segment fault. The only fault which cannot cause another fault of any type is page fault. Each frame of the concealed stack contains the PE and keys (KEYSH) of the faulting procedure as well as a fault code (to distinguish different types within each class) and a fault address, if appropriate. The stack itself is circular and must have allocated sufficient frames to handle the longest possible sequence of fault on fault that can occur in ring 0. Such a sequence might be: 

- Pointer (link) fault -> Segment fault
- Segment fault -> Page fault. 

Note that this particular sequence occurs before any software fault handler is entered. Also, the first segment fault enters ring C, so at least a five-level stack is necessary if the original link fault is to be processed correctly.

The second data base consists of four distinct fault tables, each pointed to by a PCS fault vector. Each entry in the table consists of four words of which the first three must be a CALF instruction. Only the page fault table must be locked to memory and only the ring 0 table must be in a pre-defined (SDW exists) segment (otherwise, segment fault might recurse infinitely). Naturally, the ring 0 table as well as the PCF is carefully audited by ring 0 procedures.

2. CALF

The CALF instruction has two major functions. First, to avoid holding interrupts for too long, the CALF instruction defines a restart point in fault handling since it has a PE (i.e., it is a macro-machine instruction). As a result, it is quite possible to suspend a process in the middle of getting to a software fault handler. Second, it allows a straightforward mechanism to simulate a procedure call from the faulting procedure (at the instruction causing the fault) to the fault handler.

The instruction itself is a three-word generic in which the second and third words are a 32-bit pointer to the fault handler. To simulate the procedure call, the PE and KEYS from the concealed stack are placed in the fault handler's stack frame along with the other base registers (only the PE and KEYS have been changed to point to the CALF and to enter 64V addressing mode) to be used by the standard procedure return (PRIN) instruction. In addition, the fault code and address are placed in the fault handler's stack as if they were arguments passed by a standard procedure call (PCL) instruction. After the information is moved from the concealed stack it is popped. In all other respects, CALF is identical to PCL.

3. Fault Handler

The fault handler is a u-code routine that is entered from the various fault class 'front ends' and, based on process exchange mode, either simulates a P-300 type fault (JST* through segment C fault vectors) or performs the P-40C fault protocol which includes setting up a concealed stack frame, switching to 64V mode, and determining, on the basis of
information provided by the 'front end', which fault vector to use and
setting PS to point to the proper CALF in the fault table. Figure 1
is a detailed flow chart of the fault handler and Figure 10 contains a
table of the necessary setup performed by each fault class 'front end'.
Note that for F-300 faults, the full RP is also saved in the u-code
scratch register PSWPS and the machine is inhibited for one instruction
if in Ring G.

C. Checks

Checks, unlike faults, are CPU events which are asynchronous with, and
are not caused by, normal instruction execution. Rather, they are
events which are either invisible (e.g., an ECC corrected error) or
fatal (e.g., missing memory module) to the currently executing
procedure and perhaps the CPU entirely (e.g., machine check). Checks
essentially represent processor faults as opposed to process or
procedure faults. Four check classes have been defined as follows:

<table>
<thead>
<tr>
<th>Check</th>
<th>P-400</th>
<th>P-300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Fail</td>
<td>Power Failure</td>
<td>same</td>
</tr>
<tr>
<td>Memory Parity</td>
<td>ECC corrected</td>
<td>ECC uncorrected</td>
</tr>
<tr>
<td>Machine Check</td>
<td>Fatal CPU error</td>
<td>Memory Parity</td>
</tr>
<tr>
<td>Missing Memory Module</td>
<td>Memory module does not exist</td>
<td>same</td>
</tr>
</tbody>
</table>

Unlike faults which can be stacked and interrupts which cause a process
to be suspended, each check class has a single save area (check block)
consisting of eight words in the interrupt segment (#4) in which PS
and KEYS (high and low) are saved in the first four locations (check
header) and the remaining four locations contain software code
(probably a JMP). Figure 12 is a picture of the check data base as
well as a description of the necessary u-code setup required before
going to the common check handler. In addition to the memory data
base, three 32-bit registers are used as a diagnostic status word (DSW)
to help a software check handler sort out what happened. Figure 13
shows the format of the DSW.

Check reporting (traps) is controlled by the two low order bits in the
models (KEYSL). The possible modes are:

- **MCW = 0**: no reporting
- **MCW = 1**: report memory parity (uncorrected) only
- **MCW = 2**: report unrecovered errors only
- **MCW = 3**: report all errors

The check trap can result in two possible actions depending upon the
type of check that occurred and the type of u-code which was trapped.
If the trapped code was either DMX, PIO, or external interrupt
processing (unless the error was a machine check for RCM parity), or if
the check was for an ECC corrected (ECCC) error, the
end-of-instruction flag is set, REGIV is set to the proper
offset/vector, MCW is set to 0 (except ECCC which sets it to 2), and a
code RIN to the trapped step is executed. In this way, the 10 bus is
always left in a clean state. In all other cases, the check to
software occurs immediately. Figure 14 is a detailed flow chart
showing the operation of the check trap handlers.

The common check handler is entered from various check 'front ends'
and, based on process exchange mode, either simulates a P-300 type
check (JST+ through segment 0 check vectors) or performs the P-400
check protocol which includes setting up the check header, inhibiting
the machine, and switching to 64V addressing mode. In either mode, MCW
is set to 0 before going to software. Figure 15 is a detailed flow
chart of the check handler and Figure 12 contains a table of the
necessary setup performed by each check class 'front end'.

III. REGISTER FILES

The PRIME 400/500 contains four distinct register files. Each file is
further divided into halves, each 32 locations (registers) long, and
each 16 bits wide. One half is referred to as the high half and the
other as the low half. Since both halves are addressed together, each
register file contains 32, 32-bit register or 64, 16-bit registers.
The register files, numbered from 0, are used as follows:

RF0 = u-code scratch and system registers
RF1 = 32 DMA channels
RF2 = User register set
RF3 = User register set

This layout of register files allows easy expansion to eight register
files, thus adding four new user register sets. All user register sets
have the same internal format and the DMA register file simply consists
of 32 channel registers. Channel register '20 within RF1 is equivalent
to the P-300 DMA registers '20 and '21. Channel register '22 is mapped
to '22 and '23. In this way, the mapping proceeds for each even
register in RF1 to channel register '36, mapped to '36 and '37. All
other RF1 registers represent additional DMA channels over the P-300.
Figure 16 shows the internal structure (usage) of RFU and the user
register sets (RF2, RF3). Note that all user register sets contain the
segment number of the Ready List/PCB segment (OWNERH) and a cell for
the modals (KEYSL). It is necessary, before entering process exchange
mode, to set OWNERH in ALL register sets to the proper value and to
NEVER alter it thereafter. Although all register sets contain a cell
for the modals, only the current register set (CRS) contains the valid
modals. It is therefore necessary, whenever register sets are
switched, to copy the modals into the new register set. Currently,
only the Dispatcher switches register sets. CRS is defined and
specified by the three bit field labeled 'CRS' in the modals. Since
this field can span up to eight register files, but two are used for
u-code scratch and DMA, user register sets are numbered from 2 - 7. Of
course, only 2 and 3 are currently implemented. Thus, for the
400/500, the CRS field must always have bit 9 off, bit 10 on, and bit
1 selects the register set (as if 0 and 1 were the numbers). In fact,
the u-code will only look at bit 11.
Direct register file addressing (not using CRS) is accomplished either with the LOAD/STOR instructions or via the control panel. The register files are ordered sequentially with an absolute address of 0 addressing RF0-register C (u-code scratch/system file), '40 addressing RF1-register C (DMA file), '1GC addressing RF2-register C (user set 2), and '14G addressing RF3-register C (user set 3).

Beside each register name where appropriate, is the PRIME-3CC mode mapping from address traces to registers (e.g., the X register is the high half of GR7).

IV. CONTROL PANEL

The control panel for the P-4CC/5CC is the same physical panel used for the P-1GC/2GC/3GC. It's functionality was enhanced by improving the u-code in the CP. All switches and selectors operate exactly as for the P-3CC with the exception of the sense switches in the up position.

Figure 17 is a diagram of the functionality of the switches. Notice that with all switches down, any FETCH/STORE operations are to/from memory-mapped. As long as segmentation mode is not turned on, mapped and absolute are the same, thus preserving compatibility. If SS4 down were absolute, address traps could not occur and would thus be incompatible. Notice also that SS5-16 in the up position changes meaning descending upon SS4. When mapped, all 12 switches are read as a 12-bit segment number. Then absolute SS11-16 are used as the 6 high order bits of the 22-bit physical address. To address any P-30 registers, all sense switches should be placed in the down position and addresses between 0 and '37 specified.

P-40G/5CC registers are accessed by raising SS1. Then, if SS2 is down, the low order 5 bits of the address are used to access 32-bit registers C-'37 within CRS. If SS2 is raised, the full 7 bit address is used to access any register in any register file. The addresses, as shown in Figure 16, are 0-'137=u-code scratch/system, '40-‘77=DMA, '10C-‘137=user set 2, and ‘14G-’177=user set 3. SS4 is used to access either the high half (up) or the low half (down) of the selected register. For all register accesses, the Y-1 functions will advance the register address before the access, exactly as for memory accesses. Wrap around will occur on the appropriate number of bits, since any bits of higher order are ignored for the access.

The control panel data register is TR2H and the address register is TR3. Upon entering the control panel routine, RP is saved in TR3 and (RP) is saved in TR2H. In addition, the keys (KEYSH) are updated to reflect accurately the live keys. Thereafter, TR3H is not altered by the control panel itself so RP is always remembered. However, on exit, PER is used to update RPH and KEYSH is used to update all the keys. As a result, single stepping can change segments as well as keys and modes. Figure 12 is a detailed flow chart of the control panel routine.

The only exception to the control panel entry protocol is that if a Fault, Check, or external Interrupt attempts to vector through a vector containing 0 in P-3CC mode, the following registers will contain:
RP: address of 'trapped' instruction
P9H: SN of 'trapped' instruction
KEYSH: proper keys
TR2H: (data) 0
TR3: (address) 0|0
TR2L: address, in segment 0, of the 'vector' containing 0

V. CP TIMER

Resolution = 1024 u-sec

Turned on by DISPATCHER before dispatch.

Turned off by:
WAIT after/during save
DISP before changing CRS

On tick, u-code increments the interval timer (TIMER) in RF(CRS). When that overflows, bit 16 in the PCE abort flags (memory) is set to cause a process fault.

It is the responsibility of software that resets the interval timer to maintain the elapsed timer.
Ready List: All pointers are 16-bit word number pointers within the PCB segment. The segment number is contained in the high portion of the OWNER pointer within each register set.

All PCB start addresses must be even (bit 16 = 0). The end of the ready list is marked with a BQL entry = 1.

FIGURE 1.
WAIT LIST STRUCTURE

Figure 2.
Process Control Block (PCB)

Figure 3.
Figure 4.
On Entry, RP is saved in register file

OP code Bit 16 = 0 end

I begin

Figure 5.
Note: All interrupt breaks result in a return to top of the dispatcher flag.

If valid keys are not available, use the instructions to set ID(CRS) in dispatcher flag.

Controller Timer

Figures 6.

FETCH

Process Fault

yes

no

Setup keys and Program Counter

Turn on CP + timer

Interrupts

Interrupts

ENB

allow interrupt break (insure RP and live keys are valid)

CRS = FCEA

Figure 6.

keyraf:

Cbase' All interrupt breaks result in a return to top of the dispatcher flag.

If valid keys are not available, use the instructions to set ID(CRS) in dispatcher flag.

Controller Timer

Figures 6.

FETCH

Process Fault

yes

no

Setup keys and Program Counter

Turn on CP + timer

Interrupts

Interrupts

ENB

allow interrupt break (insure RP and live keys are valid)

CRS = FCEA

Figure 6.

keyraf:
The registers to be saved are a parameter passed as a starting RF address in (TR0,L).

- Save timer and Keys
- Save mask = 0

* All registers saved?
  - No: Shift save mask left
  - Yes: RF(REAL) ≠ 0

  - No: Set bit in save mask
  - Yes: Store register into PCB

SD(CRS) = 1

RTN

Figure 7.
Figure 9.
Fault Table

```
<table>
<thead>
<tr>
<th>Fault #</th>
<th>Offset</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>#0</td>
<td>#4</td>
<td>FV0</td>
</tr>
<tr>
<td>#1</td>
<td>#4</td>
<td>FV1</td>
</tr>
<tr>
<td>#2</td>
<td>#4</td>
<td>FV2</td>
</tr>
<tr>
<td>#3</td>
<td>#4</td>
<td>FV3</td>
</tr>
<tr>
<td>#4</td>
<td>#4</td>
<td>FV4</td>
</tr>
<tr>
<td>#5</td>
<td>#4</td>
<td>FV5</td>
</tr>
<tr>
<td>#6</td>
<td>#4</td>
<td>FV6</td>
</tr>
<tr>
<td>#7</td>
<td>#4</td>
<td>FV7</td>
</tr>
</tbody>
</table>
```

Notes: Fault Vectors contain appropriate ring numbers
Ring 0 Vector address = Fault # + '62

Faults

```
Fault # | Offset | Vector |
--------|--------|--------|
0       | 0      | '62    |
1       | 0      | '63    |
2       | 10     | '64    |
3       | 14     | '65    |
4       | 20     | '66    |
5       | 40     | '72    |
6       | 44     | '73    |
7       | 50     | '74    |
8       | 54     | '75    |
9       | 60     | '76    |
10      | 64     | '77    |
```

Entry to common handler (FAULT)

```
RP = proper RP to save (backed up if necessary)
FCODEH(11) = fault code (if needed)
FADDR(12) = address (if needed)
FADDR(12) = fault offset
LATCH6 = 0 fault
LATCH7 = 0 go to ring 0
```

Figure 10.
On Entry:
RP = proper RP to save
FCODEH(11) = fault code
FCODEL = fault%4
FADDR = address(SN)
FADDRL(2) = address(WN)
LATCH6 = fault
LATCH7 = use ring 0

INPUT KEYS
Process Exchange Mode

INHI off (P300)

vector = '62+fault%'

((vector))=RPL

PSWPS :=RP

RP=0
RPL=(vector)+1

FETCH

 tá

(t1)=(LAST)
(t2)=(NEXT)

SAVE PSH, PSL KEYS, FCODE and FADDR in concealed stack
(t1)=(NEXT)+6

(t1)+(t2)=-1
(END=NEXT)

yes

(t3)=(FIRST)

no

(NEXT)=(t3)

(t3)=FV0
(t3)=FV(ring#)
(t3)=PFV

RP=((t3))+fault%4

Keys=64V

Figure 11.
Software check catchers reside in the Interrupt segment (4) and are 8 words each. The first 4 words are used as a PSW save area as:

The check offsets and corresponding P300 vectors are:

<table>
<thead>
<tr>
<th>Check</th>
<th>Offset</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Fail</td>
<td>'200</td>
<td>'60</td>
</tr>
<tr>
<td>Memory Par.</td>
<td>'270</td>
<td>'67</td>
</tr>
<tr>
<td>Machine Chk.</td>
<td>'300</td>
<td>'70</td>
</tr>
<tr>
<td>Missing Mem.</td>
<td>'310</td>
<td>'71</td>
</tr>
</tbody>
</table>

In all cases, the saved PS is the current FS when the check occurred.

Entry to common handler (CHECK)

$REJ1V = 'P400 offset$
$P300 vector=(offset-'200)$

$LATCH5 = 0$ RP is proper RP to save
= 1 proper RP is in PBSAVE
(Note: FBSAVE=0 implies in dispatcher)

Figure 12.
Diagnostic Status Word (DSW)

60 bits, Registers '34,'35,'36 (named OSWRMA, OSWSTAT, and OSWPS)

<table>
<thead>
<tr>
<th>Bits</th>
<th>OSWRMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>33,48</td>
<td>OSWSTATH Valid on all checks except Power Fail</td>
</tr>
<tr>
<td>49,64</td>
<td>OSWSTATL as follows:</td>
</tr>
<tr>
<td>65,86</td>
<td>OSWPS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine Check Code</th>
<th>C</th>
<th>M</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inv Count</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M Bus</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine Check Code</th>
<th>C</th>
<th>M</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inv Count</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M Bus</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

33: Cl=Check Immediate
34: MC=Machine Check
35: MP=Memory Parity (ECC)
36: MM=Missing Memory
37,39: Machine Check Code
0=Peripheral Data (EPD) Output
1=Peripheral Address (EPA) Input
2=Memory Data (EMO) Output
3=Cache Data (RCD)
4=Peripheral Address (EPA) Output
5=RDX-EPD Input
6=Memory Address (EMA)
7=Register File
40: Not RCM Parity (Reset for RCM Parity error - XCS only)
41: ECCU=ECC Uncorrectable Error
42: ECCC=ECC Corrected Error
43: Eup inv=RP backup count (44-46) Invalid
44,46: RP Backup Count-amount RPL (DSWPS) was incremented in current instruction
47: CMX, set if check occurred during CMX
48: IO Bus, set if check occurred during CMX, P10 or Interrupt u-code
49: RMA Inv=OSWRMA Invalid (Possible from ECCU and MM only)
50: Reserved
51,55: ECCO Syndrome=5 syndrome bits on a corrected error
56: Mod #=Low order address bit of memory module causing the error
57,58: Reserved
59,64: u-Verify test $ set on failure during Master Clear or VIRY instruction

Validity:
Always :1-33,43,47-48,55-80
If bit 34 set :37-40
If bit 35 :41-42,56 If bit 42 set:51-55
If bit 36 :56
If bit 43 reset:44-46

It is the responsibility of the check handling software to clear the DSW after a check has been processed.

Figure 13.
Figure 14.
On Entry: RP=proper RP to save
RE0IV=P400 offset
Machine Check Mode set

\[
\text{CHECK} \quad \text{INH} \quad \text{Latch}5 \quad \text{reset} \quad \text{RP=PSAVE}
\]

\[
\text{OFF} \quad \text{Process Exchange Mode} \quad \text{on} \quad (P400)
\]

\[
\text{FAULT}
\]

\[
\text{SAVE PEH, PEL, KEYS, and MODALS (before INH) in CHECK header}
\]

\[
\text{Keys=64V, INH}
\]

\[
\text{RP=4)(OFFSET+4)}
\]

\[
\text{FFETCH}
\]

*The actual calculation of P300 check vector is as follows:

In CHECK: FCODEL = OFFSET/2-110
= (OFFSET-'220)/2

In FAULT: FCODEL = (FCODEL+310)/4
= FCODEL/4+162
= ((OFFSET-'220)/2)/4+162
= (OFFSET-'220)/8+162
= OFFSET/'200-'20)/8+162
= (OFFSET-'200)/8-2+162
= (OFFSET-'200)/8+160

This circuitous calculation is used to avoid dividing a negative number on a power fail check.

Note: '200 (Power fail offset)-'220 = -'20.

Figure 15.
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLEX=0</td>
<td>Allows FLEX Faults</td>
</tr>
<tr>
<td>EV</td>
<td>Enables interrupts</td>
</tr>
<tr>
<td>CRS</td>
<td>Current Register Set</td>
</tr>
<tr>
<td>MIO</td>
<td>Mapped I/O</td>
</tr>
<tr>
<td>PXM</td>
<td>Process Exchange Mode</td>
</tr>
<tr>
<td>SEG</td>
<td>Segmentation Mode</td>
</tr>
<tr>
<td>MCM</td>
<td>Machine Check Mode</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Mode</th>
<th>FLEX=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>DEFAULT</td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>DISPATCHER</td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>SAVE DONE</td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 16:**

---

**Legend:**

- **KEYSH**: Key States
- **KEYSL**: Key States (Modals)
- **Addr**: Address
- **Mode**: Mode
- **Dispatcher**: Save Done
Notes: With all switches down, control panel works exactly as for the P-300 following either a Master Clear or a HALT if not running in segmented mode. It is necessary to make mapped memory accesses if address traps are to be generated. If running segmented, memory accesses will be mapped to segment 0 unless an explicit segment number is entered in SS5-16.

Registers: Register address is in address register (switches down).
- For CRS, only low order 5 bits are used; for absolute, only low order 8 bits are used.
- Y+1 (STORE/FETCH) operates exactly as for memory with the address being pre-incremented.

Null Vector: In P-300 mode, if an external interrupt, fault, or check attempts to vector through a memory location containing a 0, the following action is taken:

- HALT
- data and address lights cleared
- RP = address trapped
- PSH = RPH
- TR2L = address of vector

Figure 17.